Docket No. 42390.P18121
Express Mail No. EV339917366US

UNITED STATES PATENT APPLICATION

FOR

A METHOD AND SYSTEM FOR ASSIGNING REGISTER CLASS THROUGH EFFICIENT DATAFLOW ANALYSIS

Inventors:

Bo Huang
Jinquan Dai
Cotton Seed

Prepared by:
BLAKELY SOKOLOFF TAYLOR & ZAFMAN LLP
12400 Wilshire Boulevard, Seventh Floor
Los Angeles, California 90025
(310) 207-3800

A METHOD AND SYSTEM FOR ASSIGNING REGISTER CLASS THROUGH EFFICIENT DATAFLOW ANALYSIS

BACKGROUND

Field

[0001] The embodiments relate to compiler technology, and more particularly to compiler optimization through register class assignment.

Description of the Related Art

[0002] When designing and implementing compilers, register allocation and register assignment are very important in increasing a computer program's efficiency. Register allocation determines the program values that should be stored in a machine's (e.g., a computer system) registers at each program point instead of being stored in the memory. Register assignment determines which register each allocated program value should be located. Compiler developers sometimes ignore register class assignment. The reason why register class assignment is usually ignored is because most machines contain very few register classes. Typically, most machines only have two sorts of registers: integer and floating-point registers.

[0003] For those machines having two register classes (i.e., integer and floating point), each register operand of an instruction requires a fixed register class, i.e. the register class assignment is explicitly performed during the instruction set architecture (ISA) design. In view of modern computer architecture design, however, register class assignment becomes increasingly difficult to determine. For example, in some advanced processors abundant register classes are available. And, an instruction register operand is allowed to be assigned with a physical register in different register classes.

[0004] To make the problem more complicated, the register class assignment can not be performed randomly. Only specified register classes can be applied to a register operand for a specified instruction. **Figures 1A** and **1B** illustrate an example that demonstrates flexible register class assignment to an arithmetic logical unit (ALU) instruction. **Figure 1A** illustrates an example of an ALU instruction. **Figure 1B** illustrates an example of possible register class assignment to symbolic registers A_OP and B_OP. This kind of

complexity makes the compiler design and implementation more and more challenging.

BRIEF DESCRIPTION OF THE DRAWINGS

[0005] The embodiments of the invention are illustrated by way of example and not by way of limitation in the figures of the accompanying drawings in which like references indicate similar elements. It should be noted that references to "an" embodiment of the invention in this disclosure are not necessarily to the same embodiment, and they mean at least one.

[0006] **Figure 1A** illustrates an example of an ALU instruction.

[0007] **Figure 1B** illustrates an example of possible register class assignment to the two symbolic registers in the instruction described in Figure 1A.

[0008] **Figure 2** illustrates an intermediate representation of a target machine instruction.

[0009] **Figure 3** illustrates the definition of the register class assignment map.

[0010] **Figure 4** illustrates a process of an embodiment for register class assignment.

[0011] **Figure 5** illustrates pseudo code of an embodiment for initial register class assignment.

[0012] **Figure 6** illustrates an example of inter-block register class fixup of an embodiment.

[0013] **Figure 7** illustrates pseudo code of an embodiment for calculating OUT_M(i).

[0014] **Figure 8** illustrates pseudo code of an embodiment for a linear time dataflow framework.

[0015] **Figure 9A** illustrates an example of an embodiment for hosting register class fixups.

[0016] **Figure 9B** illustrates an example of an embodiment for sinking register class fixups.

[0017] **Figure 10** illustrates a block diagram of a system of one embodiment.

[0018] **Figure 11** illustrates a representative computer in conjunction with which embodiments may be practiced.

DETAILED DESCRIPTION

[0019] The Embodiments discussed herein generally relate to a method and system for assigning register class through efficient dataflow analysis. Referring to the figures, exemplary embodiments will now be described. The exemplary embodiments are provided to illustrate the embodiments and should not be construed as limiting the scope of the embodiments.

[0020] Reference in the specification to "an embodiment," "one embodiment," "some embodiments," or "other embodiments" means that a particular feature, structure, or characteristic described in connection with the embodiments is included in at least some embodiments, but not necessarily all embodiments. The various appearances "an embodiment," "one embodiment," or "some embodiments" are not necessarily all referring to the same embodiments. If the specification states a component, feature, structure, or characteristic "may", "might", or "could" be included, that particular component, feature, structure, or characteristic is not required to be included. If the specification or claim refers to "a" or "an" element, that does not mean there is only one of the element. If the specification or claims refer to "an additional" element, that does not preclude there being more than one of the additional element.

[0021] In the case where a target machine has m register classes C_1, C_2, ... C_m, each register class C_1 contains N_i physical registers (i.e. $N_i = |C_i|$). In addition, the intermediate representation of each instruction of target machine looks like the format depicted in **Figure 2**, in which Dest, Src_1 and Src_2 might be symbolic registers. In one embodiment by, using an efficient

dataflow analysis for each compilation unit (function), assignment of a register class to each symbolic register is included in each instruction.

[0022] In another embodiment, for a symbolic register s to be included in an instruction *i*, if register class C_k is assigned to s in *i*, the information of register class assignment is represented as RegClass (s, i) = C_k and symbolized as s:C_k in instruction *i*.

[0023] In one embodiment a Universal Register Class (the notation is simplified as C) is a pseudo register class that indicates the union of each register class from C_1 to C_m. A symbolic register can be assigned to C in a process where register class assignment is performed. It should be noted that sumbolic registers should be assigned with an actually existing register class (C_1, C_2, ... C_m) after the register class assignment process is completed. From the definition, the following two equations always hold for each i that satisfies $1 \leq i \leq m$.

[0024] $C_i \cap C = C_i$

$C_i \cup C = C$

[0025] In one embodiment a Register Class Array (the notation is simplified as A) is a constant array with $\sum_{1 \leq i \leq m} N_i$ elements, the value of each element is defined as follows:

$$A(i) = \begin{cases} C_1 (1 \leq i \leq N_1) \\ C_2 (N_1 + 1 \leq i \leq N_1 + N_2) \\ \cdots \\ C_m (\sum_{1 \leq j \leq m-1} N_j + 1 \leq i \leq \sum_{1 \leq j \leq m} N_j) \end{cases}$$

[0026] A global integer, A_Index, is used to iterate the array in the register class assignment process. In one embodiment, during the initialization portion of the assignment process, A_Index is initialized to 1.

[0027] In one embodiment a Register Class Assignment Map (the notation is simplified as M) is a map used to track the register class assignment

to each symbolic register. Each element of M has the form of (s, c), which means that register class c is assigned to the symbolic register s. Assuming that a compilation unit (function) contains N_s symbolic registers, the register class assignment map that contains N_s elements can be defined as illustrated in **Figure 3**.

[0028] In one embodiment a Register Class Fixup is an instruction inserted to when register class assignments are modified. For example, if symbolic register s is first assigned with register class C_1 and in later analysis it is determined to assign s with C_2, then an additional instruction is added to move $s:C_1$ to $s:C_2$ ($s:C_2 \leftarrow s:C_1$). The additional instruction move is called register class fixup from C_1 to C_2. In one embodiment register class fixup might be inserted either before one instruction or after one instruction, or both, providing that the semantic of the program is unchanged. Since more than one symbolic register might appear in one instruction, it's possible that several register class fixups are needed to accommodate the moves.

[0029] **Figure 4** illustrates a process of one embodiment where a register class is assigned to each symbolic register and the assignment is included in each instruction. Process 400 begins with block 410 where an initial register class assignment is made. For some instructions, it is required that only a specific register class can be assigned to the symbolic register operand that appears in those instructions (either destination operand or source operand). While for other symbolic registers that do not have such requirement, universal register class **C** is assigned to the symbolic register operand. **Figure 5** illustrates pseudo code of an algorithm included in one embodiment for initial register class assignment.

[0030] After block 410 is completed, process 400 continues with block 420. In block 420, register class assignment is performed through conjunctive forward dataflow analysis. For an arbitrary basic block b, the register class assignment map is assigned at the block entry as IN_M(b), while the register class assignment map at the block exit is assigned as OUT_M(b). The similar notion can be applied to an instruction, i.e., IN_M(i) and OUT_M(i), which are used to represent the register class assignment map at the entry of instruction ***i*** and at the exit of instruction ***i*** respectively. The following equations and

algorithms are used to calculate OUT_M(b), IN_M(b), IN_M(i) and OUT_M(i). As for OUT_M(b), it can be calculated by using the following equation: OUT_M(b)=OUT_M(i), where i is the last instruction inside basic block b:

$$IN\_M(b)= \bigcap_{p \in Pred(b)} OUT\_M(p)$$

The "intersection" of register class assignment maps is defined as follows:

$$M_1 \cap M_2 \cap \ldots \cap M_n = \{(s_i, c_{1si} \cap c_{2si} \cap \ldots \cap c_{nsi}) \mid 1 \le i \le N_s\}$$

Because $C_i \cap C = C_i$, if any $c_{ks}(1 \le k \le n)$ equals C, the "intersection" with it can be ignored. Based on this, the value of $c_{1si} \cap \ldots \cap c_{nsi}$ depends on those $c_{ksi}(1 \le k \le n)$ while $c_{ksi} \neq C$. If the number of C_j's appearing in $c_{1si} \cap c_{2si} \cap \ldots \cap c_{nsi}$ is the largest one, the following equation holds in the algorithm $c_{1si} \cap c_{2si} \cap \ldots \cap c_{nsi} = C_j$. For example:

$$C_2 \cap C_5 \cap C_2 \cap C \cap C_3 = C_2 \cap C_5 \cap C_2 \cap C_3 = C_2$$

If the appearance of each C_j is the same in $c_{1si} \cap c_{2si} \cap \ldots \cap c_{nsi}$, then $c_{1si} \cap c_{2si} \cap \ldots \cap c_{nsi}$ equals to the first C_j for the condition of $C_j \neq C$. This equation reflects the following principle: If different register classes are assigned to a symbolic register in a different path, then in the joint point it is more beneficial to choose the register class assignment that are valid in most paths ("register class fixup"). In one embodiment register class fixup is introduced at the end of each proceeding basic block where the register class assignment is different from the assignment in the joint point. If a symbolic register has been assigned with C in one proceeding basic block, register class fixup for that symbolic register is not necessary at the end of that basic block. **Figure 6** illustrates an example for inter-block register class fixup.

[0031] In one embodiment, if b has no proceeding basic block, i.e. b is the entry block of a compilation unit (function), then IN_M(b)=INIT_M, while INIT_M is a special register class assignment map where each symbolic register is assigned with C, the universal register class. IN_M(i)=OUT_M(p), where p is the proceeding instruction of i inside basic block b. If i is the first instruction of basic block b, then IN_M(i)=IN_M(b).

[0032] In one embodiment two function interfaces are used in the algorithm. The first function is "BOOL IsValidRegClassAssignment (Inst, NthOperand, RegClass)." This function returns TRUE if the register class assignment (RegClass) to the NthOperand of instruction Inst is valid. Otherwise, this function returns FALSE.

[0033] The second function interface is "REGISTERCLASS GetNextRegClass(Inst, NthOperand)." This function is used to find the correct register class assignment for the NthOperand of instruction Inst. In one embodiment pseudo code for GetNextRegClass is as follows:

```
while (TRUE)
    If(IsValidRegClassAssignment (Inst, NthOperand, A(A_Index)))
        break;
    Suppose A(A_Index) equals Cj, then reassign A_Index to be ( ∑_{1≤i≤j} Ni)%( ∑_{1≤i≤m} Ni)+1
ReturnRegClass = A(A_Index);
A_Index = (A_Index%( ∑_{1≤i≤m} Ni))+1;
return ReturnRegClass.
```

[0034] It should be noted that because there must exist one register class that satisfies the register class assignment for the NthOperand of instruction Inst, the while loop won't become an infinite loop. Actually, the mechanism used to adjust A_Index can guarantee that no more than N_i register classes are assigned to the adjacently used N_i symbolic registers, which helps reduce the interferences in the future physical register assignment. According to the definition of Register Class Array A, assigining $(\sum_{1\le i\le j} N_i)\%(\sum_{1\le i\le m} N_i)+1$ to A_Index can guarantee that C_1 will be the next register class assignment candidate after C_m.

[0035] **Figure 7** illustrates pseudo code of an algorithm used in one embodiment for calculating OUT_M(i). With the above data flow equations, a conjunctive forward dataflow analysis can be performed to calculate the

register class assignment for each symbolic register inside each instruction. The dataflow analysis is efficient because each basic block is iterated only once.

[0036] **Figure 8** illustrates pseudo code for an algorithm for the linear time dataflow framework. During the process of constructing the topological order of the dataflow graph, algorithms should be used to break the strong connected component to make the topological sort applicable.

[0037] After block420 is completed, process 400 continues with block 430. Block 430 reduces the register class fixups through code hoisting/sinking and dead code elimination. It should be noted that dead code elimination is a technique for improving the efficiency of a program by eliminating certain unnecessary code that may be either completely or partially dead. Completely dead code is code whose computed values are never used. Partially dead code has its computed values used along some paths but not others. Hoisting or lifting of the instruction to a different location may improve efficiency if the hoisting can be performed without executing additional instructions along any other path. Sinking or moving of instructions can improve efficiency if the sinking can be performed without executing additional instructions along any other path.

[0038] Because many register class fixups might be performed in block 420, the optimization of code hoisting/sinking can be performed to hoist/sink the fixups so that fewer register class fixups will be kept. **Figure 9A-B** illustrates hoisting and sinking register class fixups. In one embodiment a dead code removal pass can be performed to remove unnecessary register class fixups. Using the case illustrated in **Figure 6** as an example, the fixup can be removed if the first appearance of **s** in the joint block is a destination operand.

[0039] Upon completion of block 430, process 400 continues with block 440. Block 440 performs register renaming to ensure the uniqueness of register class assignment. After block 430, a symbolic register might be assigned with various register classes in different instructions. To ensure that each symbolic register is assigned with only one register class, a register renaming pass is performed in block 440. For example, if a symbolic register is assigned with K different register classes, K different symbolic registers will be used instead,

with each symbolic register being assigned with only one register class. Their corresponding appearances are replaced with the renamed symbolic registers.

[0040] By using the above described embodiments the problem of register class assignment is efficiently resolved. These embodiments provide a general framework for performing register class assignment through dataflow analysis. These embodiments are of particular benefit for modern processors having abundant register classes. The complete data flow analysis is a linear one because it iterates each basic block only once. Through using the register class array (A) and a process for adjusting the array index (A_Inder), the register class assignment algorithm achieves the result of having no more than N_i register classes being assigned to the adjacently used N_i symbolic registers. This helps reduce interferences in future physical register assignment.

[0041] **Figure 10** illustrates a block diagram of a compiler coupled to a processor for an embodiment having a register class assignment process according to the above-mentioned embodiments. In particular, **Figure 10** illustrates processor 1020, compiler 1010 and register set 1030. In one embodiment, compiler 1010 is a computer program on a machine (i.e., a compiler program) that can reside on a secondary storage medium (e.g., a hard drive on a machine) and is executed on processor 1020. Those of ordinary skill in the art will appreciate that other components can be coupled to processor 1020 (e.g., cache memory, local memory, memory controller, etc.), not shown in **Figure 10**; only those parts necessary to describe the invention in an enabling manner are provided.

[0042] Processor 1020 has a number of registers ($R_1 - R_N$) in register set 1030. In this embodiment, N can be 1 or more. Compiler 1010 is a compiler that can execute in processor 1020 in which a source program is inputted, causing compiler 1010 to generate an executable program, as is well known in the art. Compiler 1010 can also execute on a target machine as well. It is important to appreciate that the embodiments are not limited to any particular type of source program, as the type of computer programming languages used to write the source program may vary from procedural code type languages to object oriented languages. In one embodiment, the executable program is a set of assembly code instructions, as is well known in the art.

[0043] It should be noted that compiler 1010 can be run on either a host or target machine to generate executable code. The host and target machines can have different architectures. The compiled code can run on the host, target machine or other machines completely separate or networked. In one embodiment, while the compiler runs on the host machine, the compiled code is only run on a target machine. In one embodiment, the host and target machines are identical. In another embodiment, the host and target machines have different architectures from one another.

[0044] **Figure 11** illustrates a diagram of a representative host machine 1110 and target machine 1120 in conjunction with which embodiments may be practiced. It is noted that embodiments of the invention may be practiced on other electronic devices, including but not limited to, a set-top box connected to the Internet, notebook computers, personal digital assistants (PDAs), palm personal computers (PCs), notebooks, servers, workstations, etc. In one embodiment host machine 1110 and target machine 1120 are computer devices, such as a desktop personal computer (PC), server, workstation, etc. In one embodiment host machine 1110 is operatively coupled to a monitor, a user interface (e.g., a keyboard) and a pointing device (not shown). In another embodiment target machine 1120 is operatively coupled to monitor, user interface (e.g., a keyboard) and pointing device (not shown). Host machine 1110 includes at least one processor 1125 (e.g., one of a number of Intel® processors, such as Pentium ®), local memory 1127, such as random-access memory (RAM), static random access memory (SRAM), dynamic random access memory (DRAM), synchronous DRAM (SDRAM), read-only memory (ROM), etc., and one or more storage devices 1127, such as a hard disk drive, a floppy disk drive (into which a floppy disk can be inserted), an optical disk drive, a tape cartridge drive, etc. The memory, hard drives, floppy disks, etc., are types of computer-readable media.

[0045] In one embodiment host machine 1110 includes compiler 1115 and register set 1130. In one embodiment, compiler 1115 is a computer program (i.e., a compiler program) that can reside on a secondary storage medium (e.g., a hard drive on a machine, RAM, etc.) and is executed on a processor, such as processor 1125. Those of ordinary skill in the art will

appreciate that other components can be coupled to processors 1125 and 1135 (e.g., cache memory, local memory, memory controller, etc.), not shown in **Figure 11**; only those parts necessary to describe the invention in an enabling manner are provided. In one embodiment, compiler 1115 runs on processor 1125 of host machine 1110, performs optimization, assigns register classes and creates compiled code. In one embodiment, the compiled code runs on target machine 1120 and not on host machine 1110. In another embodiment, compiled code created on host machine 1110 runs on host machine 1110. It should be noted that when compiled code is created on host machine 1110, it can run on other target machines (not shown) that have a different architecture than host machine 1110.

[0046] In one embodiment processor 1125 has a number of registers (R_1 – R_N) in register set 1130. In this embodiment, N can be 1 or more. In one embodiment processor 1135 has a number of registers (R_1 – R_M) in register set 1140. In this embodiment, M can be 1 or more. Compiler 1115 is a compiler that can execute in a processor in which a source program is inputted, causing the compiler to generate an executable program, as is well known in the art. Compiler 1115 can execute on a target machine such as target machine 1120 or a host machine, such as host machine 1110. It is important to appreciate that the embodiments are not limited to any particular type of source program, as the type of computer programming languages used to write the source program may vary from procedural code type languages to object oriented languages. In one embodiment, the executable program is a set of assembly code instructions, as is well known in the art.

[0047] The embodiments are not particularly limited to any type of host machine 1110 or target machines 1120. Residing on host machine 1110 is a computer readable medium storing a computer program that is executed on machine 1110 or target machine 1120. Compiler optimization is performed by the computer program in accordance with the embodiments. In one embodiment, target machine has compiler 1116. It should be noted that in other embodiments, compiler 1116 is not included or required. Compiler 1116 can be completely different from compiler 1115. In another embodiment, compiler 1115 and compiler 1116 can be similar or identical.

[0048] The above embodiments can also be stored on a device or machine-readable medium and be read by a machine to perform instructions. The machine-readable medium includes any mechanism that provides (i.e., stores and/or transmits) information in a form readable by a machine (e.g., a computer). For example, a machine-readable medium includes read-only memory (ROM); random-access memory (RAM); magnetic disk storage media; optical storage media; flash memory devices; biological electrical, mechanical systems; electrical, optical, acoustical or other form of propagated signals (e.g., carrier waves, infrared signals, digital signals, etc.). The device or machine-readable medium may include a micro-electromechanical system (MEMS), nanotechnology devices, organic, holographic, solid-state memory device and/or a rotating magnetic or optical disk. The device or machine-readable medium may be distributed when partitions of instructions have been separated into different machines, such as across an interconnection of computers.

[0049] While certain exemplary embodiments have been described and shown in the accompanying drawings, it is to be understood that such embodiments are merely illustrative of and not restrictive on the broad invention, and that this invention not be limited to the specific constructions and arrangements shown and described, since various other modifications may occur to those ordinarily skilled in the art.